2.4 Defining Milner's EXP Language and W Type Inferencer

Milner defined and proved the correctness of his W type inferencer in the context of a simple higher-order language that he called EXP [1]. Recall that W is the basis for the type checkers of all statically typed functional languages, including ML, OCAML, HASKELL, etc. EXP is a simple expression language containing lambda abstraction and application, conditional, fix point, and "let" and "letrec" binders. Here is the annotated K syntax of EXP (the application is strict in both its subexpressions, while the conditional is strict only in its first subexpression; also, let and letrec are defined as syntactic sugar):

K-Annotated Syntax of EXP

Var	::=	standard identifiers	
Exp	::=	*Var* | ... add basic values (Booleans, integers, etc.)	
	|	λ *Var* . *Exp*	
	|	*Exp Exp*	[*strict*]
	|	μ *Var* . *Exp*	
	|	if *Exp* then *Exp* else *Exp*	[*strict*(1)]
	|	let *Var* = *Exp* in *Exp*	[let $x = e$ in $e' = (\lambda x.e')\, e$]
	|	letrec *Var Var* = *Exp* in *Exp*	[letrec $f\ x = e$ in $e' =$ let $f = \mu f.(\lambda x.e)$ in e']

Below is a straightforward K semantics of EXP with a one-item configuration.

K Configuration and Semantics of EXP

			$[\![e]\!] = [\![k(e)]\!]$
Val	::=	λ *Var* . *Exp* | ...(Booleans, integers, etc.)	$[\![k(v)]\!] = v$
KResult	::=	*Val*	$k((\lambda x.e)\, v\rangle = k(e[x \leftarrow v]\rangle$
KProper	::=	μ *Var* . *Exp*	$k(\mu\ x.e\rangle = k(e[x \leftarrow \mu\ x.e]\rangle$
ConfigItem	::=	$k(K)$	if true then e_1 else $e_2 \rightarrow e_1$
Config	::=	*Val* | $[\![K]\!]$ | $[\![Set[ConfigItem]]\!]$	if false then e_1 else $e_2 \rightarrow e_2$

To test the semantics, one can now execute programs like the factorial:

```
letrec f x = if x <= 0 then 1 else x * f(x - 1)
in f (f 5)
```

Our definition yields, when run in Maude, the following result in 12,312 rewrites and about 0.2 seconds:

```
668950291344912705758811805409037258675274633313802981029567135230163355724496298936687416527198498
130815763789321409055253440858940812185989848111438965000596496052125696000000000000000000000000000000
```

We next define the W type inferencer in [1] using the same K approach. The K annotated syntax changes the conditional to strict in all its arguments, and desugars let making it strict in its second argument:

K-Annotated Syntax of EXP for W

Var	::=	standard identifiers	
Exp	::=	*Var* | ... add basic values (Booleans, integers, etc.)	
	|	λ *Var* . *Exp*	
	|	*Exp Exp*	[*strict*]
	|	μ *Var* . *Exp*	
	|	if *Exp* then *Exp* else *Exp*	[*strict*]
	|	let *Var* = *Exp* in *Exp*	[*strict*(2)]
	|	letrec *Var Var* = *Exp* in *Exp*	[letrec $f\ x = e$ in $e' =$ let $f = \mu f.(\lambda x.e)$ in e']

Unification over type expressions is needed. Fortunately, unification is straightforward to define equationally using set matching; we define it using rewrite rules, though, to emphasize that it is executable ($t_v \in \mathit{TypeVar}$):

Unification

$$
\begin{array}{rcl}
\mathit{Type} & ::= & \ldots \mid \mathit{int} \mid \mathit{bool} \mid \mathit{Type} \mapsto \mathit{Type} \mid \mathit{TypeVar} \\
\mathit{Eqn} & ::= & \mathit{Type} = \mathit{Type} \\
\mathit{Eqns} & ::= & \mathit{Set}[\mathit{Eqn}]
\end{array}
$$

$$(t = t) \rightarrow \cdot$$

$$(t_1 \mapsto t_2 = t'_1 \mapsto t'_2) \rightarrow (t_1 = t'_1),\ (t_2 = t'_2)$$

$$(t = t_v) \rightarrow (t_v = t) \quad \text{when } t \notin \mathit{TypeVar}$$

$$t_v = t,\ t_v = t' \rightarrow t_v = t,\ t = t' \quad \text{when } t, t' \neq t_v$$

$$t_v = t,\ t'_v = t' \rightarrow t_v = t,\ t'_v = t'[t_v \leftarrow t] \quad \text{when } t_v \neq t'_v,\ t_v \neq t,\ t'_v \neq t',\ \text{and } t_v \in \mathit{vars}(t')$$

The first rule above eliminates non-informative type equalities. The second distributes equalities over function types to equalities over their sources and their targets; the third swaps type equalities for convenience (to always have type variables as lhs's of equalities); the fourth ensures that, eventually, no two type equalities have the same lhs variable; finally, the fifth rule canonizes the substitution. As expected, the rules above take a set of type equalities and eventually produce a most general unifier for them:

Theorem 2 *Let $\gamma \in \mathit{Eqns}$ be a set of type equations. Then:*

- *The five-rule rewrite system above terminates (modulo AC); let $\theta \in \mathit{Eqns}$ be the normal form of γ.*
- *γ is unifiable iff θ contains only pairs of the form $t_v = t$, where $t_v \notin \mathit{vars}(t)$; if that is the case, then we identify θ with the implicit substitution that it comprises, that is, $\theta(t_v) = t$ when there is some type equality $t_v = t$ in θ, and $\theta(t_v) = t_v$ when there is no type equality of the form $t_v = t$ in θ.*
- *If γ is unifiable then θ is idempotent (i.e., $\theta \circ \theta = \theta$) and is a most general unifier of γ.*

Therefore, the five rules above give us a simple rewriting procedure for unification. Experiments show that it is also quite efficient (our resulting type inferencer is comparable in performance with state of the art implementations of type inference in languages like OCAML). The structure of θ in the second item above may be expensive to check every time the unification procedure is invoked; in our Maude implementation of the rules above, we sub-sort (once and for all) each equality of the form $t_v = t$ with $t_v \notin \mathit{vars}(t)$ to a "proper" equality, and then allow only proper equalities in the sort *Eqns* (the improper ones remain part of the "kind" [*Eqns*]). If $\gamma \in \mathit{Eqns}$ is a set of type equations and $t \in \mathit{Type}$ is some type expression, then we let $\gamma[t]$ denote $\theta(t)$; if γ is not unifiable, then $\gamma[t]$ is some error term (in the kind [*Type*] when using Maude).

Below is the K definition of W. The configuration has four items: the computation, the type environment, the set of type equations (constraints), and a counter for generating fresh type variables. Due to the strictness attributes, we can assume that the corresponding arguments of the language constructs (in which these constructs were defined strict) have already been "evaluated" to their types and the corresponding type constraints have been propagated. Lambda and fix-point abstractions perform normal bindings in the type environment, while the let performs a special binding, namely one to a type wrapped with a new "*let*" type construct. When names are looked up in the type environment, the "*let*" types are instantiated with fresh type variables for their "universal" type variables, namely those that do not occur in the type environment.

K Configuration and Semantics of W

$$
\begin{array}{rcl}
\mathit{KResult} & ::= & \mathit{Type} \\
\mathit{TEnv} & ::= & \mathit{Map}[\mathit{Name}, \mathit{Type}] \\
\mathit{Type} & ::= & ... \mid \mathit{let}(\mathit{Type}) \\
\mathit{ConfigItem} & ::= & k(K) \mid \mathit{tenv}(\mathit{TEnv}) \mid \mathit{eqns}(\mathit{Eqns}) \mid \mathit{nextType}(\mathit{TypeVar}) \\
\mathit{Config} & ::= & \mathit{Type} \mid [\![K]\!] \mid [\![\mathit{Set}[\mathit{ConfigItem}]]\!]
\end{array}
$$

$[\![e]\!] = [\![k(e)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t_v)]\!]$

$[\![\langle k(t)\ \mathit{eqns}(\gamma)\rangle]\!] = \gamma[t]$

$i \to \mathit{int},\ \mathit{true} \to \mathit{bool},\ \mathit{false} \to \mathit{bool},$ (and similarly for all the other desired basic values)

$k(\underset{\mathit{int}}{\underline{t_1 + t_2}}\rangle\ \mathit{eqns}\langle\underset{t_1 = \mathit{int},\ t_2 = \mathit{int}}{\underline{\quad\cdot\quad}}\rangle$ (and similarly for all the other standard operators)

$k(\underset{(\gamma[t])[tl \leftarrow tl']}{\underline{\quad x \quad}}\rangle\ \mathit{tenv}(\eta)\ \mathit{eqns}(\gamma)\ \mathit{nextType}(\underset{t_v + |tl|}{\underline{\quad t_v \quad}})$ when $\eta[x] = \mathit{let}(t),\ tl = \mathit{vars}(\gamma[t]) - \mathit{vars}(\eta)$ and $tl' = t_v \ldots (t_v + |tl| - 1)$

$k(\underset{\eta[x]}{\underline{\ x\ }}\rangle\ \mathit{tenv}(\eta)$ when $\eta[x] \neq \mathit{let}(t)$

$k(\underset{(t_v \to e) \curvearrowright \mathit{restore}(\eta)}{\underline{\quad \lambda x.e \quad}}\rangle\ \mathit{tenv}(\underset{\eta[x \leftarrow t_v]}{\underline{\quad \eta \quad}})\ \mathit{nextType}(\underset{t_v + 1}{\underline{\quad t_v \quad}})$

$K ::= \cdots \mid \mathit{Type} \to K \quad [\mathit{strict}(2)]$

$k(\underset{t_v}{\underline{t_1\ t_2}}\rangle\ \mathit{eqns}\langle\underset{t_1 = t_2 \to t_v}{\underline{\quad\cdot\quad}}\rangle\ \mathit{nextType}(\underset{t_v + 1}{\underline{\quad t_v \quad}})$

$k(\underset{e \curvearrowright ?_{=}(t_v) \curvearrowright \mathit{restore}(\eta)}{\underline{\quad \mu x.e \quad}}\rangle\ \mathit{tenv}(\underset{\eta[x \leftarrow t_v]}{\underline{\quad \eta \quad}})\ \mathit{nextType}(\underset{t_v + 1}{\underline{\quad t_v \quad}})$

$k(t \to \underset{\cdot}{\underline{?_{=}t_v}}\rangle\ \mathit{eqns}\langle\underset{t_v = t}{\underline{\quad\cdot\quad}}\rangle$

$k(\underset{e \curvearrowright \mathit{restore}(\eta)}{\underline{\ \mathsf{let}\ x = t\ \mathsf{in}\ e\ }}\rangle\ \mathit{env}(\underset{\eta[x \leftarrow \mathit{let}(t)]}{\underline{\quad \eta \quad}})$

$k(\underset{t_1}{\underline{\mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2}}\rangle\ \mathit{eqns}\langle\underset{t = \mathit{bool},\ t_1 = t_2}{\underline{\quad\cdot\quad}}\rangle$

We believe that the K definition above of W is as simple, if not simpler, to understand as the original W procedure proposed by Milner in [1]. However, note that the procedure in [1] is an *algorithm*, almost an *implementation*, rather than a formal, logic-based definition! Our K definition above is nothing but an ordinary rewrite logic theory, same as the formal definition of EXP itself. That should not, and indeed it does not, mean that our K definition, when executed, must necessarily be slower than an actual implementation of W. Experiments using Maude show that our K definition above of W is comparable or even outperforms state of the art implementations of type inference in conventional functional languages. For example, on our experiments it was faster than the type inferencers of SML and Haskell, and only about twice slower than that of OCAML. Concretely, the program (which is polymorphic in $2^n + 1$ type variables!)

$$
\begin{array}{l}
\mathsf{let}\ f_0 = \lambda x.\lambda y.x\ \mathsf{in} \\
\quad \mathsf{let}\ f_1 = \lambda x.f_0(f_0 x)\ \mathsf{in} \\
\quad\quad \mathsf{let}\ f_2 = \lambda x.f_1(f_1 x)\ \mathsf{in} \\
\quad\quad\quad ... \\
\quad\quad\quad\quad \mathsf{let}\ f_n = \lambda x.f_{n-1}(f_{n-1} x)\ \mathsf{in}\ f_n
\end{array}
$$

takes the following time/space resources to be type checked using OCAML, Haskell, SML, and our K definition executed in Maude, respectively:

-	n = 10		n = 12		n = 13		n = 14	
OCAML (version 3.09.3)	0.6s	3M	8.1s	5M	31.2s	8M	120.6s	13M
Haskell (ghci version 6.7.20070312)	2.7s	25M	42.4s	31M	207.8s	38 M	1177s	61M
SML (version 110.59)	5s	76M	110.5s	324M	2129.2s	950M	out of M	
W in K/Maude2.2 with memo	1.2s	17M	18.9s	65M	78s	191M	304s	653M
W in K/Maude2.2 without memo	2.2s	10M	22.4s	47M	81s	152M	305s	563M

The experiments above have been conducted on a 3.4GHz/2GB Linux machine. Only the user time has been recorded. Except for SML, the user time was very close to the real time; for SML, the real time was 30% larger than the user time. These ratios appear to scale and be preserved for other programs, too. Moreover, extensions of the type system with lists, products, side effects (through referencing, dereferencing and assignment) and weak polymorphism did not add any noticable slowdown. Therefore, our K definitions surprisingly yield quite realistic *implementations* of type checkers/inferencers when executed on efficient rewrite engines! While calculating the numbers above, Maude run at an average of 3 million rewrites per second. In Maude, memoization can be enabled by adding the attribute "`[memo]`" to operations whose results one wants memoized; in our case, we only experimented with memoizing the "built-in" operation `vars : Type -> Set{TypeVar}` in `k-prelude.maude`, which extracts the set of type variables that occur in a type. Memoization appears to pay off when the polymorphic types are small, which is typically the case.

Our Maude "implementation" of an extension[1] of the K definition of W above has about 30 lines of code. How is it be possible that a formal definition of a type system that one can write in 30 lines of code can be executed *as is* more efficiently than well-engineered implementations of the same type system in widely used programming languages? We think that the answer to this question involves at least two aspects. On the one hand, Maude, despite its generality, is itself a well-engineered rewrite engine implementing state-of-the-art AC matching and term indexing algorithms. On the other hand, our K definition makes intensive use of what Maude is very good at, namely AC matching. For example, note the fourth rule in our rewrite definition of unification[2] that precedes Theorem 2: the type variable t_v appears twice in the lhs of the rule, once in each of the two type equalities involved. Maude will therefore need to search and then index for two type equalities in the set of type constraints which share the same type variable. Similarly, but even more complicately, the fifth rule involves two type equalities, the second containing in its t' some occurence of the type variable t_v that appears in the first. Without appropriate indexing to avoid rematching of rules, which is what Maude does well, such operations can be very expensive. Moreover, note that our type constraints can be "solved" incrementally (by applying the five unification rewrite rules), as generated, into a most general substitution; incremental solving of the type constraints can have a significant impact on the complexity of unification as we defined it, and Maude indeed does that (one can see it by tracing/logging Maude's rewrite steps).

[1] With conventional arithmetic and boolean operators added for writing and testing our definition on meaningful programs

[2] Type unification is "built-in" in K: it is defined as shown above in `k-prelude.maude`.